E190AR: Final Project Report

Charles Dawson & Peter Johnson

***Abstract*— This report describes our approach to the E190AR final project, including vision-based steering for the racing challenge and autonomous path planning and navigation for the maze challenge. For the former challenge, we employed a novel control scheme in which the arUco marker is attached to the driver, allowing them to use intuitive driving movements to control the robot using the laptop as a steering wheel. For the latter challenge, we use a rapidly-exploring random tree (RRT) with a stochastic path smoothing modification to produce easily navigable paths through the maze, and employ a refined point-tracking control algorithm designed to reduce collision probabilities and odometry errors to follow that path to the goal. Additionally, this report describes our attempts to implement a particle filter to improve robot state estimation, although our particle filter was not functional when we eventually navigated the maze.**

I. Introduction

In our final project, we were assigned two primary challenges. The first was a race in which we needed to manually control our romi pololu robot (see Fig. 1)using a vision-based steering system, navigating through the hallways of the Parsons Engineering Building on HMC's campus. The second challenge was an autonomous navigation task in which we were given the layout of a maze and had to design a system of ROS nodes so that our robot could dynamically generate and follow a collision-free path through the maze. To solve these challenges, we drew on our work throughout E190AR to develop system architectures capable of these vision-based-control and autonomous-navigation tasks, allowing us to eventually complete both tasks successfully. The first section describes our approach to the vision-based manual control task for the robot race, and the second section discusses our solution to the autonomous navigation task.

II. Vision-based Manual Control

To accomplish the vision-based steering task, we attached an arUco marker to the driver's chest and had them hold a laptop facing the arUco marker, so that both of the driver's hands would be free for holding the laptop. As shown in the system architecture schematic in Fig. 2, the laptop runs an `/aruco_detect` node that publishes the `/fiducial_transforms` topic giving a list of heterogeneous transformations from the laptop frame to each of the arUco markers it detects in the laptop camera's field of view (in our application, only the driver's marker should be visible). The `/fiducial_transforms` topic is subscribed to by our `/mario_kart` node, which converts

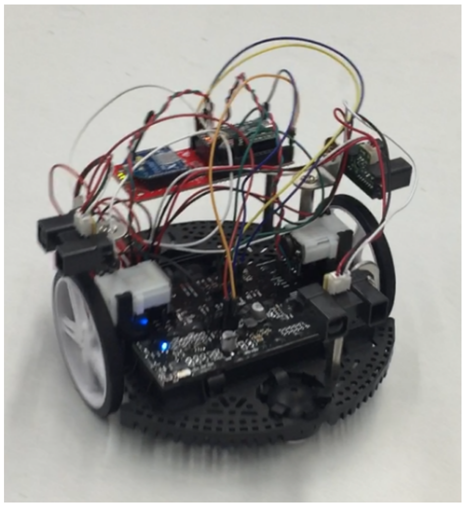
Fig. 1: E190 AR romi pololu robotics platform

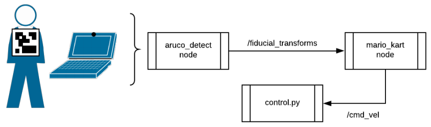

Fig. 2: System architecture for the vision-based racing control task

the relative position and orientation of the marker to control inputs, which are sent to the `/control` node via the `/cmd_vel` topic.

To convert the relative position and orientation of the arUco marker into control inputs for our robot, we developed the control scheme shown in Fig. 3. In this control scheme, the linear velocity of the robot was set to $v = 2(z - 0.5)$, where z is the distance from the laptop to the marker along the camera's z-axis in meters, and the angular velocity was set to $\omega = 1.5\phi$, where ϕ is the second Euler angle of the marker relative to the laptop frame (pitch). Additionally, a 10 cm wide dead-band was applied to the linear velocity control and a 0.2 radian (11 degree) dead-band was applied to the angular velocity control to allow the driver to easily bring the robot to a complete stop. Using this control scheme we were able to successfully navigate the racing course through Parsons Engineering Building, avoiding major obstacles and completing the course in under 2 minutes.

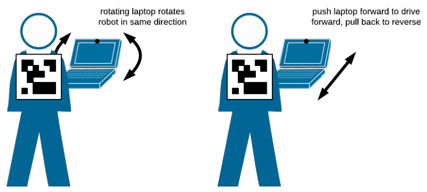
rotating laptop rotates robot in same direction
push laptop forward to drive forward, pull back to reverse

Fig. 3: Control scheme for the vision-based racing control task

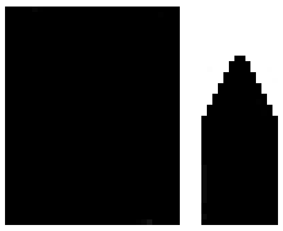

Fig. 4: Unmodified map for the autonomous navigation task

III. Autonomous Navigation

This section describes our approach to the autonomous navigation task. This task involved navigating the map shown in Fig. 4. To allow us to plan a path with the assumption of a point-like robot, we dilated this map by 3 pixels (approximately the length of the robot's body) to yield the modified map shown in Fig. 5 which we used for all of our navigation. Our decision to enlarge the map by more than just a radius was an attempt to keep the robot further away from the walls.

A. ROS System Architecture

To complete the autonomous navigation task, we developed the ROS node system architecture shown in Fig. 6. Originally, we intended to include a particle filter for estimating the pose of our robot, but due to time constraints we were not able to fully implement

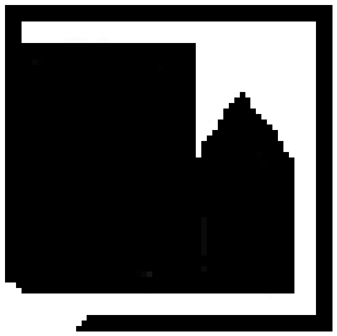

Fig. 5: Modified map for the autonomous navigation task, dilated by 3 pixels compared to the original map.

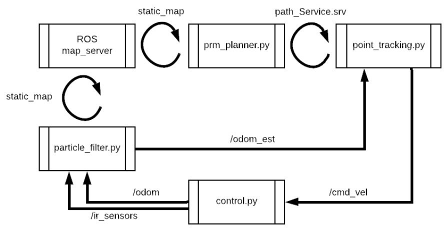

Fig. 6: Autonomous navigation ROS node architecture.

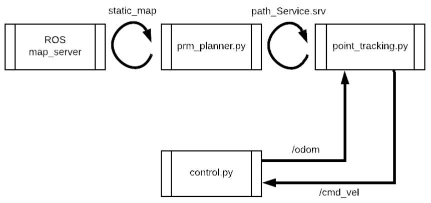

Fig. 7: Autonomous navigation ROS node architecture without particle filter.

this feature, instead falling back on the architecture shown in Fig. 7. We made three major improvements to our autonomous navigation architecture over what was developed for the labs in E190AR. These improvements were: implementing a stochastic path smoothing algorithm, modifications to the point-tracking controller, and modifications designed to reduce the error in odometry-based state estimates.

B. Path Generation

For our final project, we used the single-query rapidly-exploring random tree (SQ-RRT) path planning algorithm developed in Lab 5 as the foundation for our autonomous path planning node. However, we observed that the paths generated by the SQ-RRT algorithm tended to be inefficient and contain many unnecessary twists and turns, as shown in Fig. 8. To smooth these probabilistic paths, we employed the following stochastic path smoothing algorithm, inspired by [2]:

1) Given a path (a list of nodes in configuration space) and a parameter N, repeat N times:
2) Randomly sample two nodes from the path.
3) If those nodes are within a set distance threshold (0.4 m in our implementation) and a collision-free straight line path connects those nodes, remove all nodes between the two sampled nodes from the path, and repeat.
4) Otherwise, do not modify the path, and repeat.

This algorithm was very effective at producing smoothed paths, although we found that a fairly large number of iterations was required to increase the probability of producing a sufficiently smooth path (on the order of $N = 100$(number of nodes in original path), which is not large enough to cause a significant loss of

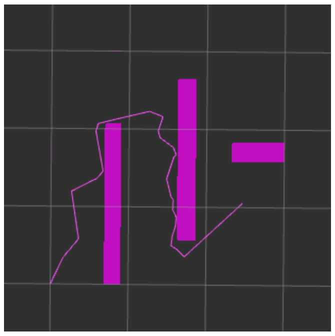

Fig. 8: Paths generated using a single-query rapidly-exploring random tree (SQ-RRT) algorithm with no path smoothing step. Note how the paths are very inefficient, containing many unnecessary twists and turns.

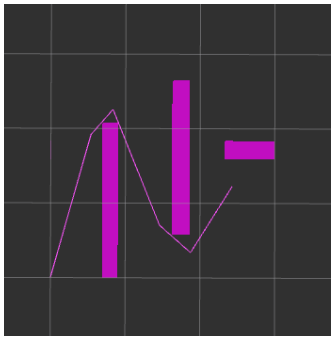

Fig. 9: A path generated using the SQ-RRT path planning algorithm with an additional stochastic path smoothing post-processing step, resulting in a much more direct path than that shown in Fig. 8.

performance). An example of the smoothed version of the path shown in Fig. 8 is given in Fig. 9.

In addition, we also modified the path planning node that we created in Lab 5 to make it act as a ROS service, allowing the point-tracking node to request the next point along its path and dynamically regenerate paths as the robot proceeds towards its goal.

C. Point Tracking

The point tracking originally developed in Lab 3 performed reasonably well for navigating the robot towards a target point in free space; however, it was not able to track along straight lines towards its goal but rather followed curved paths. Since our path-planning algorithm ensures only that straight line segments between points on the path are collision-free and makes no guarantees regarding curved paths, we had to modify our point-tracking controller to track trajectories that more closely approximated straight lines.

To do this, we replaced the single-step P-controller developed in Lab 3 with a two-step controller that first controls the robot's angular velocity according to $\omega = k_\beta \beta + k_\alpha \alpha$, where k_β and k_α are controller constants and α and β are defined in Fig. 10. Once the angle of the robot with respect to its goal (α) has been regulated to within approximately 30 degrees, the robot begins the second step where it resumes control using the P-controller developed in Lab 3, but since it begins facing more directly towards its target the robot is able to pursue a much straighter path.

This controller significantly improved our robot's ability to follow collision-free paths through the maze, since there were many instances in our eventual successful completion of the maze when a curved path would have resulted in a collision but a straight-line path remained safe.

D. Odometry Error Reduction

In Lab 2, we implemented functionality to estimate the robot's pose using odometry based on wheel encoder measurements. The Teensy microcontroller on our robot sends encoder measurements to the control node via the XBee at a rate of 2 Hz. The encoders have a resolution of 1440 count/revolution, so the left and right encoder readings ($\Delta\mathrm{Enc_L}$ and $\Delta\mathrm{Enc_R}$, measured by subtracting the previous encoder reading from the current reading) can be converted to linear distances travelled by the the left and right wheel according to the formulas

$$\Delta s_R = \Delta\mathrm{Enc_R}\frac{2\pi \mathrm{r}}{1440} \tag{1}$$

$$\Delta s_L = \Delta\mathrm{Enc_L}\frac{2\pi \mathrm{r}}{1440} \tag{2}$$

From these wheel translations, we can compute the incremental translation Δs and rotation θ of the robot body, shown in Fig. 11, using the equations

$$\Delta s = \frac{\Delta s_r + \Delta s_L}{2} \tag{3}$$

$$\Delta\theta = \frac{\Delta s_r - \Delta s_L}{2L} \tag{4}$$

$$\tag{5}$$

However, pose estimation based on odometry measurements alone is notoriously unreliable due to the accumulation of unmodelled errors and noise in the estimate. In most practical applications, odometry would be combined with a more sophisticated state estimation paradigm (e.g. Kalman or particle filtering), but due to the time constraints facing our project we took an alternative approach to reducing the error inherent in our odometry measurements. We observed that the two largest causes of accumulated error in odometry measurements are unmodelled errors such as wheel slip or the error due to the linearizing assumptions used when deriving the odometry update equations. Furthermore, we observed that the magnitude of both of these effects increase with the speed of the robot: wheel slip becomes more common at higher speeds, and the linearizing assumption that the robot moves through sequential pure rotates and pure translation becomes more accurate as the distance travelled between time steps decreases.

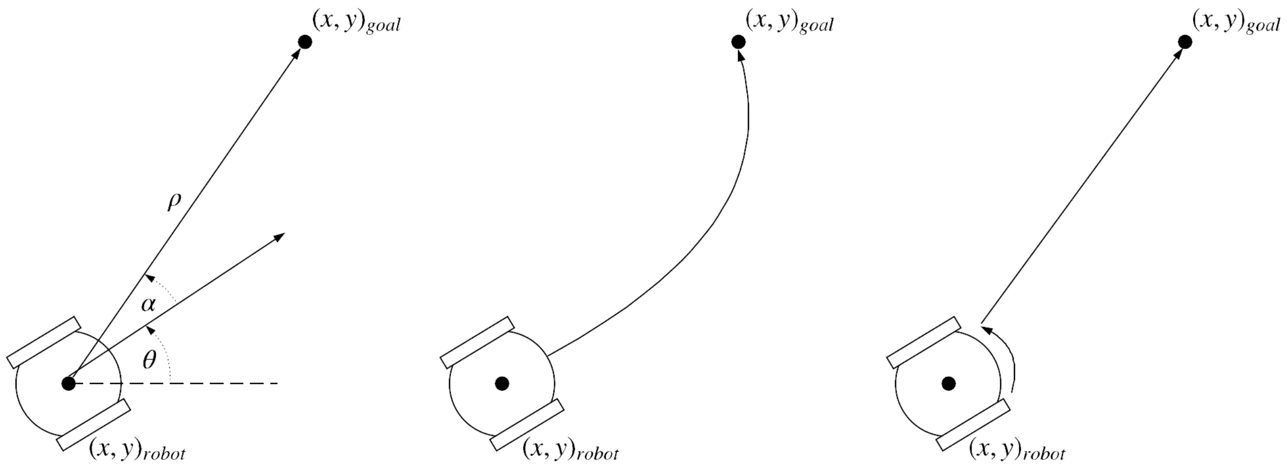

Fig. 10: (Left) A definition of the point-tracking variables of interest. (Center) Typical performance of the single-step P-controller developed for Lab 3. (Right) Typical performance of the two-step P-controller developed for the final project.

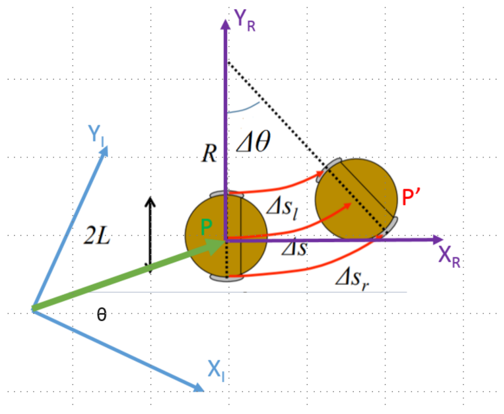

Fig. 11: Incremental wheel odometry update [?].

As a result of these observations, we determined that reducing the speed of our robot would allow us to obtain odometry measurements that were sufficiently accurate over the relatively short duration of the maze navigation task. To implement this, we modified the point tracking code to reduce the calculated linear velocity of the robot by a factor of 5 (the translational velocity was left unmodified to further improve the straight-line trajectory tracking of the robot). Although this reduced speed meant that it took our robot much longer to complete the course, it made errors due to mis-estimating the robot's pose less common. Using this modification, along with those discussed above, we were able to successfully navigate the maze, even without the particle filter state estimation functionality discussed in the next section.

E. State Estimation

Although our strategy of slowing down the robot to reduce accumulated odometry error worked for our application, in any application requiring extended operation, some measure of feedback must be incorporated into the state estimate in order to allow the robot to correct for accumulated error in its state estimation. The most commonly-used class of tools for this application are recursive Bayesian state estimators such as Extended Kalman filters or particle filters. For this project, we attempted to implement a particle filter for our robot, using measurements from IR range sensors to update the state predictions made using odometry. A particle filter works by maintaining a population of points in configuration space called particles. At each time step, the odometry measurements from the wheel encoders are used to predict the new state of each particle, moving it to a new location in configuration space (adding some random noise to the motion of each particle). When new sensor measurements are received, the filter performs an updating step by resampling the population of particles according to how likely it is that each particle would have observed the received sensor measurements [4].

We implemented a particle filter as a ROS node using Python, as pictured in the system architecture in Fig. 6. However, we encountered several bugs in our implementation that we did not have time to resolve, and so for our final demonstration we did not use a particle filter and instead relied on the state estimation produced by odometry measurements.

IV. Results & Conclusion

Although we encountered several errors on the day of the final project demonstration, largely involving errors the interaction between the path planning node and the map, our robot successfully navigated the maze less than

15 minutes after class ended on the day of the final project demonstration. Although we were not able to get our particle filter implementation working in time for the demonstration, we nevertheless were able to rely on the state estimation derived from odometry measurements for the relatively short duration of the course. If a longer navigation were needed, or if any sort of simultaneous localization and mapping behavior were desired, it would become much more important to implement a particle filter, even though odometry was sufficient for this particular navigation task. Additionally, further work could be done to improve the reliability of the path planner, ensuring that it behaves correctly when asked to plan a path from a starting point that is very close to an occupied area on the map. Despite these areas where further improvement is possible, we were still able to eventually complete both the racing and autonomous navigation challenges successfully.

The code for our final project is included in our GitHub repository, which can be found at `https://github.com/dawsonc/e190ar`

References

[1] Y. Chang. E190AR. Lab Instructions, Topic: "PRM." Department of Engineering, Harvey Mudd College, Claremont, CA.

[2] D. Touretzky, E. Tira-Thompson. 15-494 Cognitive Robotics. Class Notes, Topic: "Path Planning." Carnegie Mellon, Pittsburgh, PA.

[3] "Rapidly-exploring random tree," Wikipedia, 16-Feb-2019. [Accessed: 10-Apr-2019].

[4] C. M. Clark, "Labs," E160 - Autonomous Robot Navigation. [Online]. Available: http://www.hmc.edu/lair/E160/labs.html. [Accessed: 05-May-2019].